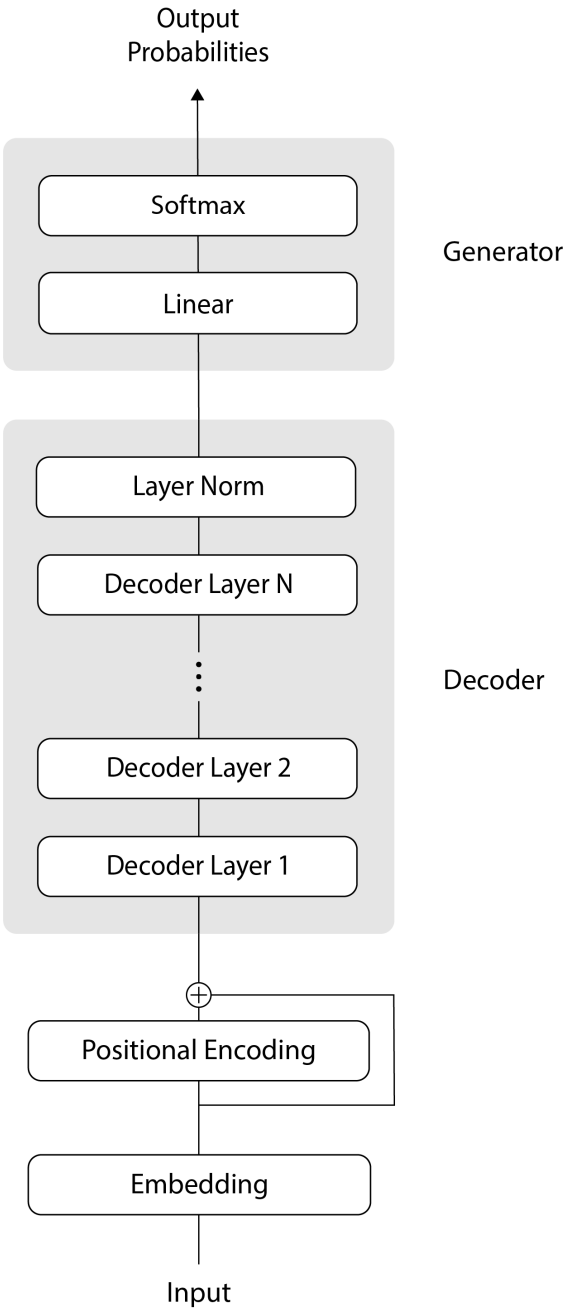
Transformer architecture of GPT model



In this diagram, the data flows from the bottom to the top, as is traditional in Transformer illustrations. Initially, our input tokens undergo a couple of encoding steps: they're encoded using an Embedding layer, followed by a Positional Encoding layer, and then the two encodings are added together.

Next, our encoded inputs go through a sequence of *N* decoding steps, followed by a normalization layer. And finally, we send our decoded data through a linear layer and a softmax, ending up with a probability distribution that we can use to select the next token.

In the sections that follow, we'll take a closer look at each of the components in this architecture.

Embedding

The Embedding layer turns each token in the input sequence into a vector of length `d_model`. The input of the Transformer consists of batches of sequences of tokens, and has shape `(batch_size, seq_len)`. The Embedding layer takes each token, which is a single number, calculates its embedding, which is a sequence of numbers of length `d_model`, and returns a tensor containing each embedding in place of the corresponding original token. Therefore, the output of this layer has shape `(batch_size, seq_len, d_model)`.

```
import torch.nn as nn

class Embeddings(nn.Module):
        def __init__(self, d_model, vocab_size):
                super(Embeddings, self).__init__()
                self.lut = nn.Embedding(vocab_size, d_model)
                self.d_model = d_model
                # input x: (batch_size, seq_len)
                # output: (batch_size, seq_len, d_model)
        def forward(self, x):
                out = self.lut(x) * math.sqrt(self.d_model)
                return out
```

The purpose of using an embedding instead of the original token is to ensure that we have a similar mathematical vector representation for tokens that are semantically similar. For example, let's consider the words "she" and

“her”. These words are semantically similar, in the sense that they both refer to a woman or girl, but the corresponding tokens can be completely different (for example, when using OpenAI’s `tiktoken` tokenizer, “she” corresponds to token 7091, and “her” corresponds to token 372). The embeddings for these two tokens will start out being very different from one another as well, because the weights of the embedding layer are initialized randomly and learned during training. But if the two words frequently appear nearby in the training data, eventually the embedding representations will converge to be similar.

Poitional Encoding

The Positional Encoding layer adds information about the absolute position and relative distance of each token in the sequence. Unlike recurrent neural networks (RNNs) or convolutional neural networks (CNNs), Transformers don’t inherently possess any notion of where in the sequence each token appears. Therefore, to capture the order of tokens in the sequence, Transformers rely on a Positional Encoding.

There are many ways to encode the positions of tokens. For example, we could implement the Positional Encoding layer by using another embedding module (similar to the previous layer), if we pass the position of each token rather than the value of each token as input. Once again, we would start with the weights in this embedding chosen randomly. Then during the training phase, the weights would learn to capture the position of each token.

```
class PositionalEncoding(nn.Module):
        def __init__(self, d_model, dropout, max_len=5000):
                super(PositionalEncoding, self).__init__()
                self.dropout = nn.Dropout(p=dropout)
                pe = torch.zeros(max_len, d_model)
                position = torch.arange(0, max_len).unsqueeze(1) #
(max_len, 1)
```

```
                div_term = torch.exp( torch.arange(0, d_model, 2) *
-(math.log(10000.0) / d_model)) # (d_model/2)
                pe[:, 0::2] = torch.sin(position * div_term) #
(max_len, d_model)
                pe[:, 1::2] = torch.cos(position * div_term) #
(max_len, d_model)
                pe = pe.unsqueeze(0) # (1, max_len, d_model)
                self.register_buffer('pe', pe)

        # input x: (batch_size, seq_len, d_model)
        # output: (batch_size, seq_len, d_model)
        def forward(self, x):
                x = x + Variable(self.pe[:, :x.size(1)],
requires_grad=False)
                return self.dropout(x)
```

Decoder

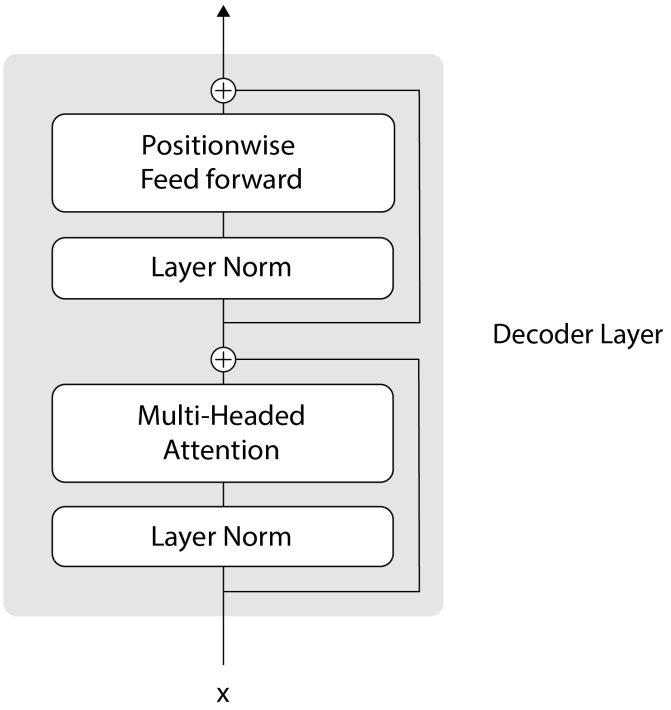
As we saw in the diagrammatic overview of the Transformer architecture, the next stage after the Embedding and Positional Encoding layers is the Decoder module. The Decoder consists of *N* copies of a Decoder Layer followed by a Layer Norm. Here's the `Decoder` class, which takes a single `DecoderLayer` instance as input to the class initializer:

```
class Decoder(nn.Module):
        def __init__(self, layer, N):
                super(Decoder, self).__init__()
                self.layers = clones(layer, N)
                self.norm = LayerNorm(layer.size)
        def forward(self, x, mask):
                for layer in self.layers:
                        x = layer(x, mask)
                return self.norm(x)
```

The Layer Norm takes an input of shape `(batch_size, seq_len, d_model)` and normalizes it over its last dimension. As a result of this step, each embedding distribution will start out as unit normal (centered around zero and with standard deviation of one). Then during training, the distribution will change shape as the parameters `a_2` and `b_2` are optimized for our scenario.

```
class LayerNorm(nn.Module):
        def __init__(self, features, eps=1e-6):
                super(LayerNorm, self).__init__()
                self.a_2 = nn.Parameter(torch.ones(features))
                self.b_2 = nn.Parameter(torch.zeros(features))
                self.eps = eps
        def forward(self, x):
                mean = x.mean(-1, keepdim=True)
                std = x.std(-1, keepdim=True)
                return self.a_2 * (x - mean) / (std + self.eps) +
self.b_2
```

The `DecoderLayer` class that we clone has the following architecture:



Here's the corresponding code:

```
class DecoderLayer(nn.Module):
        def __init__(self, size, self_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout),
2)
        def forward(self, x, mask):
                x = self.sublayer[0](x, lambda x: self.self_attn(x,
x, x, mask))
                return self.sublayer[1](x, self.feed_forward)
```

At a high level, a `DecoderLayer` consists of two main steps: the attention step, which is responsible for the communication between tokens, and the feed forward step, which is responsible for the computation of the predicted

tokens. Surrounding each of those steps, we have residual (or skip) connections, which are represented by the plus signs in the diagram. Residual connections provide an alternative path for the data to flow in the neural network, which allows skipping some layers. The data can flow through the layers within the residual connection, or it can go directly through the residual connection and skip the layers within it. In practice, residual connections are often used with deep neural networks, because they help the training to converge better. You can learn more about residual connections in the paper Deep residual learning for image recognition, from 2015. We implement these residual connections using the `SublayerConnection` module:

```
class SublayerConnection(nn.Module):
        def __init__(self, size, dropout):
                super(SublayerConnection, self).__init__()
                self.norm = LayerNorm(size)
                self.dropout = nn.Dropout(dropout)
        def forward(self, x, sublayer):
                return x + self.dropout(sublayer(self.norm(x)))
```

The feed-forward step is implemented using two linear layers with a Rectified Linear Unit (ReLU) activation function in between:

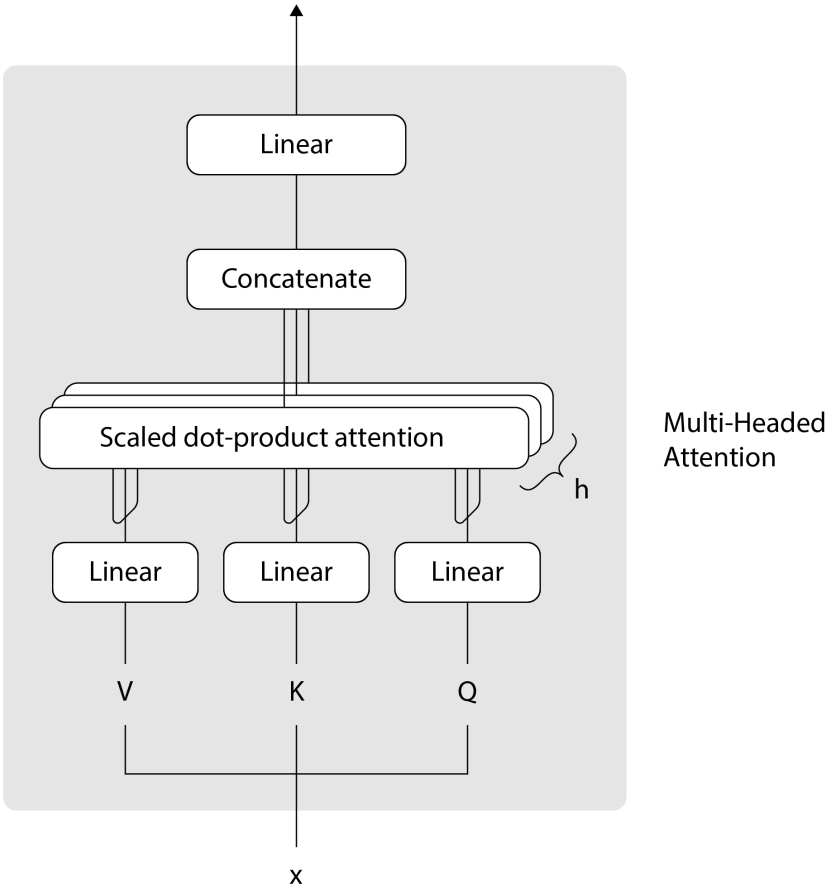
```
class PositionwiseFeedForward(nn.Module):

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
```

```
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

The attention step is the most important part of the Transformer, so we'll devote the next section to it.

Masked multi-headed self-attention

The multi-headed attention section in the previous diagram can be expanded into the following architecture:



Each multi-head attention block is made up of four consecutive levels:

- On the first level, three linear (dense) layers that each receive the queries, keys, or values
- On the second level, a scaled dot-product attention function. The operations performed on both the first and second levels are repeated *h* times and performed in parallel, according to the number of heads composing the multi-head attention block.
- On the third level, a concatenation operation that joins the outputs of the different heads
- On the fourth level, a final linear (dense) layer that produces the output

Recall as well the important components that will serve as building blocks for your implementation of the multi-head attention:

- The **queries**, **keys,** and **values**: These are the inputs to each multi-head attention block. In the encoder stage, they each carry the same input sequence after this has been embedded and augmented by positional information. Similarly, on the decoder side, the queries, keys, and values fed into the first attention block represent the same target sequence after this would have also been embedded and augmented by positional information. The second attention block of the decoder receives the encoder output in the form of keys and values, and the normalized output of the first decoder attention block as the queries. The dimensionality of the queries and keys is denoted by d(k), whereas the dimensionality of the values is denoted by d(v).
- The **projection matrices**: When applied to the queries, keys, and values, these projection matrices generate different subspace representations of each. Each attention *head* then works on one of these projected versions of the queries, keys, and values. An additional projection matrix is also applied to the output of the multi-head attention block after the outputs of each individual head would have been concatenated together. The projection matrices are learned during training.

Generator

The last step in our Transformer is the Generator, which consists of a linear layer and a softmax executed in sequence:

```
class Generator(nn.Module):
        def __init__(self, d_model, vocab):
                super(Generator, self).__init__()
                self.proj = nn.Linear(d_model, vocab)
        def forward(self, x):
                return F.log_softmax(self.proj(x), dim=-1)
```

The purpose of the linear layer is to convert the third dimension of our tensor from the internal-only `d_model` embedding dimension to the `vocab_size` dimension, which is understood by the code that calls our Transformer. The result is a tensor dimension of `(batch_size, seq_len, vocab_size)`. The purpose of the softmax is to convert the values in the third tensor dimension into a probability distribution. This tensor of probability distributions is what we return to the user.

You might remember that at the very beginning of this article, we explained that the input to the Transformer consists of batches of sequences of tokens, of shape `(batch_size, seq_len)`. And now we know that the output of the Transformer consists of batches of sequences of probability distributions, of shape `(batch_size, seq_len, vocab_size)`. Each batch contains a distribution that predicts the token that follows the first input token, another distribution that predicts the token that follows the first and second input tokens, and so on. The very last probability distribution of each batch enables us to predict the token that follows the whole input sequence, which is what we care about when doing inference.

The Generator is the last piece of our Transformer architecture, so we're ready to put it all together.

To know how to train and implement it all together , we can see the article link.

Source:

[1] https://machinelearningmastery.com/how-to-implement-multi-head-attention-from-scratch-in-tensorflow-and-keras/

[2] https://bea.stollnitz.com/blog/gpt-transformer/